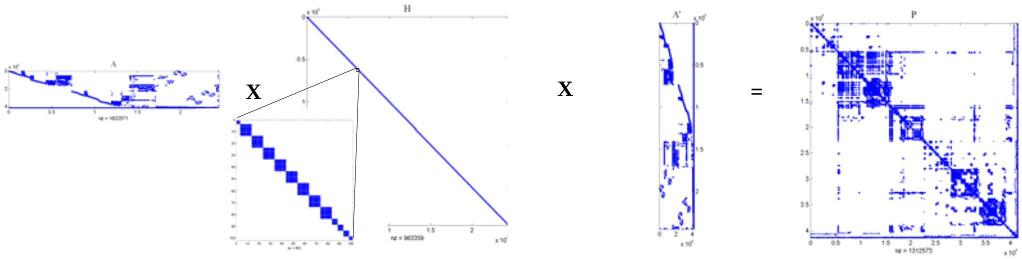
A Computationally Efficient Triple Matrix Product for a Class of Sparse Schur-complement Matrices

Eun-Jin Im, Kookmin University, Seoul, Korea, ejim@eecs.berkeley.edu
Ismail Bustany, Barcelona Design Inc., Ismail.Bustany@barcelonadesign.com
Cleve Ashcraft, Livermore Software Technology Corp., cleveashcraft@earthlink.net
James W. Demmel, U.C.Berkeley, demmel@eecs.berkeley.edu
Katherine A. Yelick, U.C.Berkeley, yelick@eecs.berkeley.edu



Berkeley Benchmarking and OPtimization Group
bebop.cs.berkeley.edu

Problem Context

- In solving a primal-dual optimization problem for a circuit design, computation of $P=AHA^t$ is repeatedly executed. (100-120 times)
- H has a symmetric block diagonal structure, $H_i = D_i + r_i r_i^t$

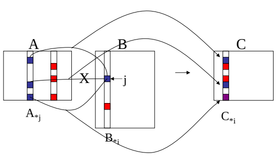
Two Implementations : One-Phase vs. Two-Phase Schemes

We compare two approaches to compute the triple-product. While one-phase scheme has an advantage over two-phase scheme by using a knowledge on the structure of matrix, the summation of sparse matrices becomes bottleneck.
Hence, we propose a row-based one-phase scheme, where the summation of sparse matrices is replaced by the summation of sparse vectors, which can be computed efficiently using a sparse accumulator.
We also improved the performance of the row-based one-phase scheme through use of additional data structures.

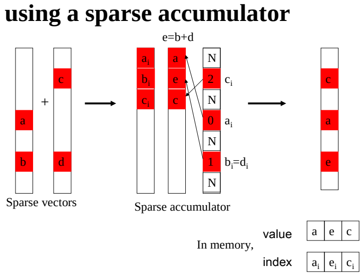
Two-Phase scheme

•P=mult(A,Q=mult(H,A^t))

In Computing C=mult(A,B)
For B_{*i} = each column of B,
For each nonzero of B_{*i}, do the following



Efficient Sparse Vector Addition using a sparse accumulator



A sparse accumulator is used in one-phase and row-based two-phase schemes.

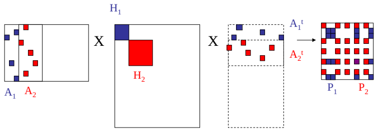
Preprocessing

•In one-phase scheme
– counting the number of nonzeros in B and P (to determine the amount of memory allocation)
– computing the structure of matrix B
– constructing row-major structure of A and B
•In two-phase scheme
– generating A^t
– counting the number of nonzeros in P and Q

One-Phase Scheme

This scheme can take advantage of known structure of H, and symmetry of P, using the following equation.

$$P = \sum_{i}^{\#\text{ of blocks in H}} (P_i = A_i H_i A_i^t) = \sum_i (A_i D_i A_i^t + (A_i r_i)(A_i r_i)^t)$$



Drawback :
a summation of sparse matrices is slow.

Row-based One-phase Scheme

Instead of adding sparse matrices, add sparse vectors for each row(column) of P.

Consider row k of P (let $B_{*i} = A_i r_i$)

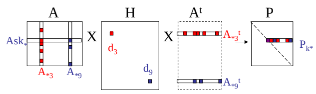
$$P_{k*} = \sum_{j}^{\#\text{ of col.s in A}} (A_{*j} d_j A_{*j}^t)_{k*} + \sum_{i}^{\#\text{ of non-unit blocks of H}} (B_{*i} B_{*i}^t)_{k*}$$
$$= \sum_j a_{kj} d_j A_{*j}^t + \sum_i b_{ki} B_{*i}^t$$
$$= \sum_{j:a_{kj}\neq 0} a_{kj} d_j A_{*j}^t + \sum_{i:b_{ki}\neq 0} b_{ki} B_{*i}^t$$

•Row-major structures of A and B are needed to access $\{j: a_{kj} != 0\}$ and $\{i: b_{ki} != 0\}$ efficiently.

Improved One-phase Scheme

$$P_{k*} = \sum_{j:a_{kj}\neq 0} a_{kj} d_j A_{*j}^t + \sum_{i:b_{ki}\neq 0} b_{ki} B_{*i}^t$$

Compute a matrix B.
Create row-major structure of A and B.
For each row(column) of P,
For j: a_{kj} != 0, do the following



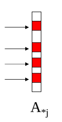
For i: b_{ki} != 0, do the similar, without scaling factor, d_i

Utilizing the Symmetry of P

$$P_{k*} = \sum_{j:a_{kj}\neq 0} a_{kj} d_j A_{*j}^t + \sum_{i:b_{ki}\neq 0} b_{ki} B_{*i}^t$$

- In computing $a_{kj} d_j A_{*j}^t$, compute $a_{kj} d_j A_{k:m,j}^t$
- by keeping an array of indices pointing to each A_{*j}'s next nonzero element, unnecessary access to A_{*j}'s is avoided.

(# of accesses to A_{*j} = # of nonzeros of A_{*j})



Performance

We predict lower and upper bounds of the execution time for one-phase and two-phase scheme using our memory model, and it is confirmed by measurements that one-phase scheme has advantage of execution time and memory over two-phase scheme.
In addition, the preprocessing cost is lower in one-phase scheme.

Memory Performance Modeling

- Memory Access
  – Dominant factor in One-phase scheme : access of elements of A in $A*H*A^t$:

$$\sum_{i}^{\#\text{ of col.s in A}} \sum_{k=1}^{nnz(A_{*i})} k + \sum_{i}^{\#\text{ of col.s in B}} \sum_{k=1}^{nnz(B_{*i})} k$$

  – Dominant factor in Two-phase scheme : access of elements of A in $A*B$:

$$\sum_{i}^{\#\text{ of col.s in A}} nnz(A_{*i}) * nnz(B_{i*})$$

- Cache Miss
  For sequentially accessed elements, spatial locality is assumed to be exploited.

- Execution Time

$$T = \alpha_1(\text{memory accesses}) + \sum_{i=1}^{k-1} (\alpha_{i+1} - \alpha_i) M_i + (\alpha_{mem} - \alpha_k) M_k$$

α_i : latency of level-i cache
α_{mem} : latency of memory
k : level of caches
M_i : cache miss in level-i cache

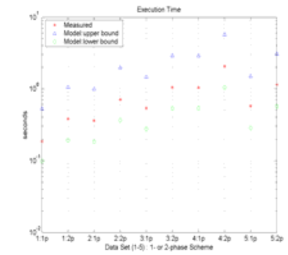
Example Matrix Set
from Circuit Design Application

	m(A)	n(A)	nnz(A)	nnz(H)		# fop.	Mem.
set 1	8648	42750	361K	195K	1-phase	11M	11M
					2-phase	24M	22M
set 2	14872	77406	667K	361K	1-phase	21M	20M
					2-phase	45M	41M
set 3	21096	112150	977K	528K	1-phase	31M	29M
					2-phase	66M	60M
set 4	39768	217030	1913K	1028K	1-phase	60M	57M
					2-phase	129M	118M
set 5	41392	244501	1633K	963K	1-phase	31M	50M
					2-phase	66M	113M

Conclusion

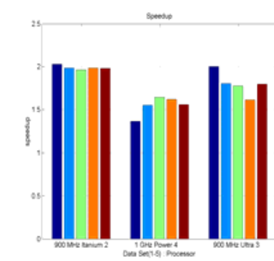
- Performance tuning of higher level sparse matrix operation than matrix-vector multiplication
- Speedup up to 2.1x
- Less than half memory requirement
- An example of algebraic transformation is used for performance tuning
- Knowledge on the special structure of the matrix is used for the algebraic transformation.

Modeled and Measured Execution Time

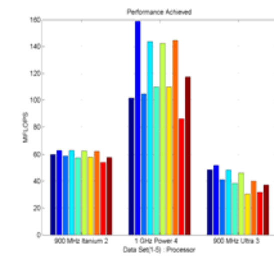


Measured Performance

Speedup



Achieved Mflop rate



Overhead of Preprocessing
relative to execution time in two-phase scheme